Segunda prova de Linguagens e Compiladores PCS-2056

Escola Politécnica da USP
PCS
Dezembro/2009
#hellweek

Leandro Cordeiro David
Wilson Faria

1a Parte da P2 de Compiladores 2009

Sintaxe em notação de Wirth

Program = {Expr}.
Expr = ("i" | Expr2).
IotaExpr = ("i" | Expr2).
quoteExp = "`" Expr Expr.
astIota = "*" IotaExpr IotaExpr.
Expr2 = "I" | "K" | "k" | "S" | "s" | NonemptyJotExpr | quoteExp |IotaExpr | "(" Expr")".
NonemptyJotExpr = ("0" | "1") { "0" | "1" } .

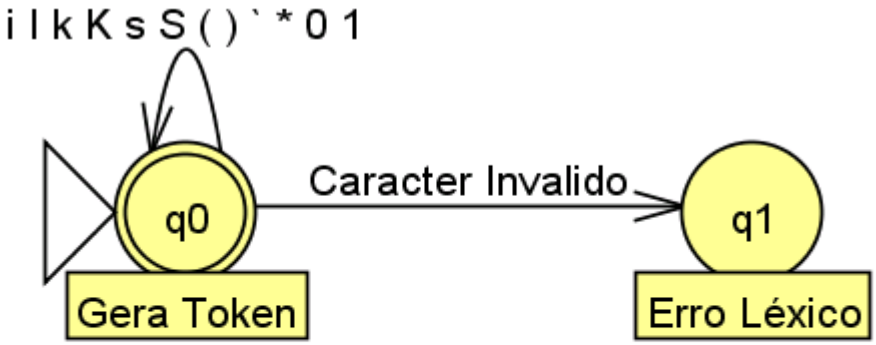
As subMaquinas quoteExp e astIot são criadas "a mais" para simplificar a análise semântica. Não preciso me preocupar em guardar , ou empilhar, a informação semântica "embutida" em "`" ou "*"

Obs. Alguns lugares deste documento mostram "Expr2" como "Expr_"

Análise Léxica

A análise léxica desta linguagem é bastante simples pois todos os terminais possuem apenas 1 caracter. A análise léxica, portanto, se resume a ler um caracter e , caso ele seja um dos terminais da linguagem, gerar um tokem, caso contrário, gerar um erro léxico.

Os terminais deta linguagem são: "i","I","K","k","S","s","`","*","0","1","(",")"



Análise sintática

Cada não terminal da notação de Wirth apresentada se materializa no compilador como uma máquina sintática. A seguir apresentamos as transições criadas para cada máquina. A seguir é apresentado para cada máquina, seu estado inicial, estados finais, transições, identificador de acao semantica e possível ação semantica a executar, bem como o correspondente automato.

As transições que têm como entrada uma maquina, empilham a maquina em execução e o próximo estado e , ao témino de execução de uma submáquina, a máquina anterior é desempilhada juntamente com o proximo estado.

Program

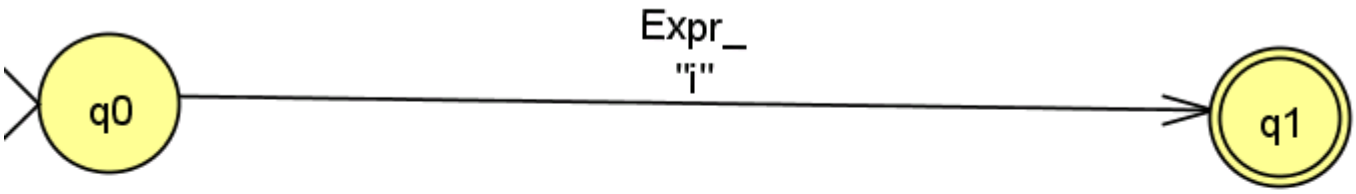
Marcação de estados:
Program = 0 { 1 Expr 2 } 1 .

```
initial: 0
final: 1
(0, Expr) -> 1 Ação semântica: Nenhuma
(1, Expr) -> 1 Ação semântica: Nenhuma
```

Expr

Marcação de estados:
Expr = 0 (0 *"i"* 2 | 0 **Expr2** 3) 1 .

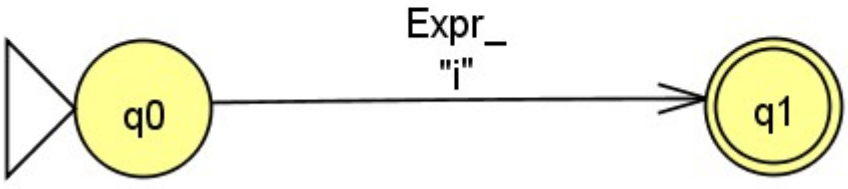
```
initial: 0
final: 1
(0, "i") -> 1  Ação semântica: gera (lambda (x) x)
(0, Expr2) -> 1  Ação semântica: Nenhuma
```



IotaExpr

Marcação de estados:
IotaExpr = 0 (0 *"i"* 2 | 0 **Expr2** 3) 1 .

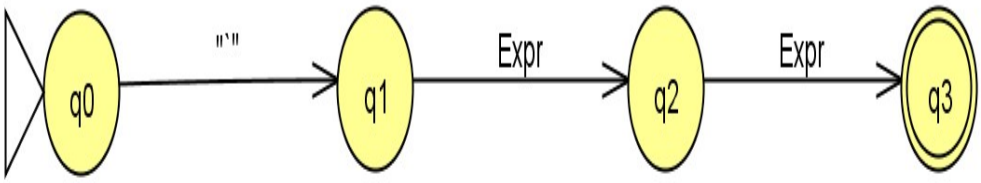
```
initial: 0
final: 1
(0, "i") -> 1  Ação semântica: gera (lambda (x) (x S K))
(0, Expr2) -> 1  Ação semântica: Nenhuma
```



quoteExp

Marcação de estados:
quoteExp = 0 ″`″ 1 **Expr** 2 **Expr** 3 .

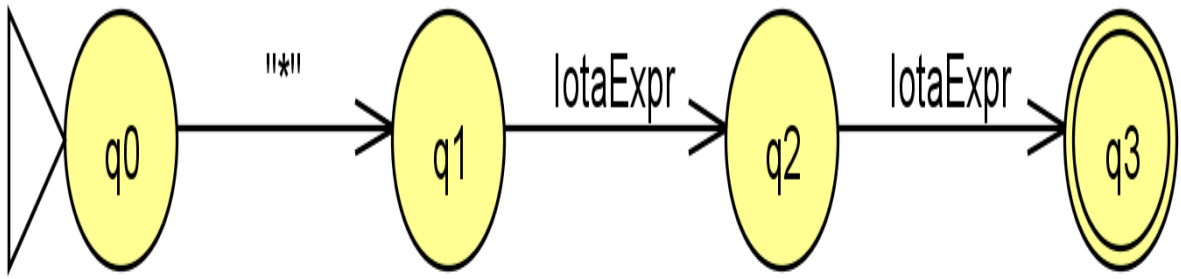
```
initial: 0
final: 3
(0, "`") -> 1  Ação semântica: Nenhuma
(1, Expr) -> 2  Ação semântica: gera Expr1
(2, Expr) -> 3  Ação semântica: gera Expr2
```



astIota

Marcação de estados:
astIota = 0 ”*” 1 **IotaExpr** 2 **IotaExpr** 3 .

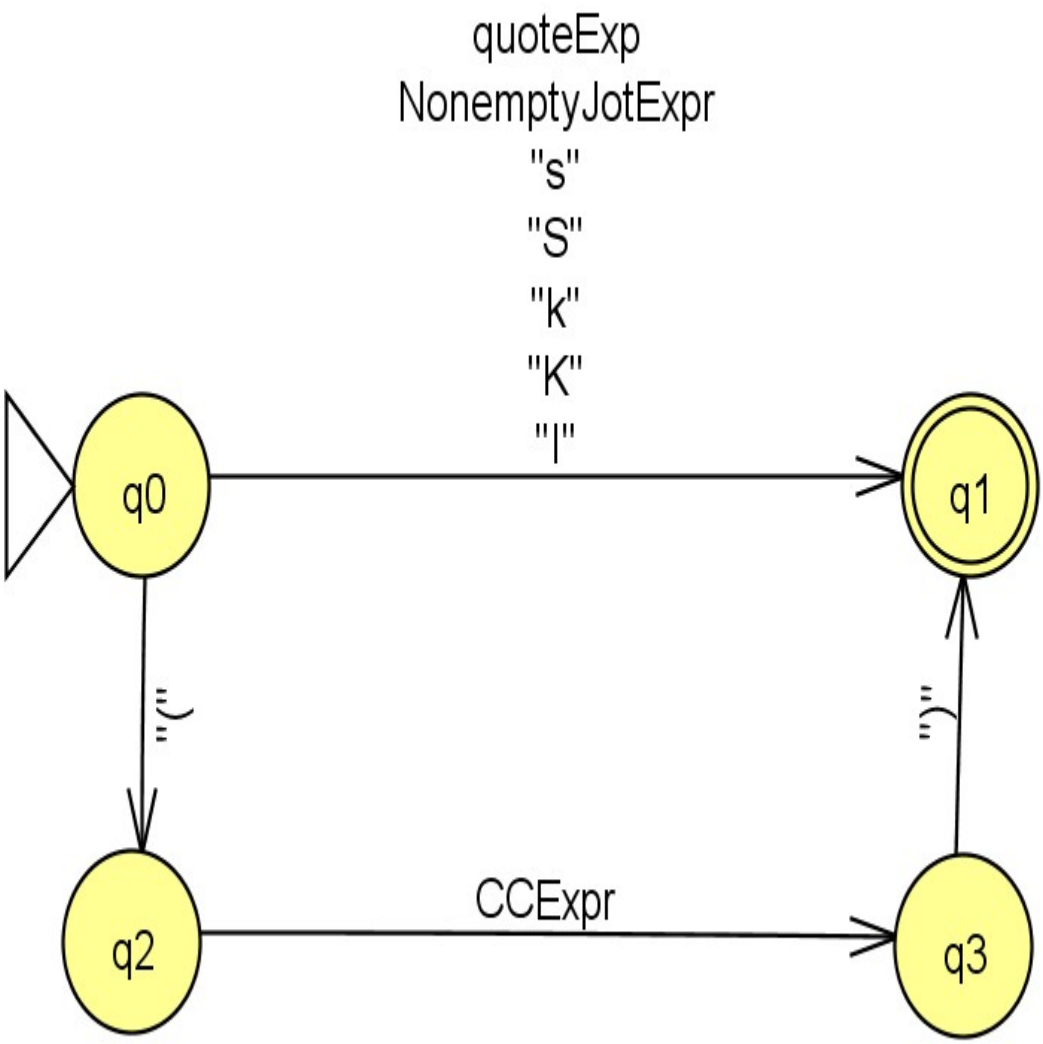
```
initial: 0
final: 3
(0, "*") -> 1  Ação semântica:
(1, IotaExpr) -> 2  Ação semântica: gera IotaExpr1
(2, IotaExpr) -> 3  Ação semântica: gera IotaExpr2
```



Expr2

Marcação de estados:
Expr2 = 0 *"I" 1* | *0 "K" 2* | *0 "k" 3* | *0 "S" 4* | *0 "s" 5* | *0* ***NonemptyJotExpr*** *6* | *0* ***quoteExp*** *7* | *0* | *0 "(" 8* ***CCExpr*** *9 ")"* 10 .

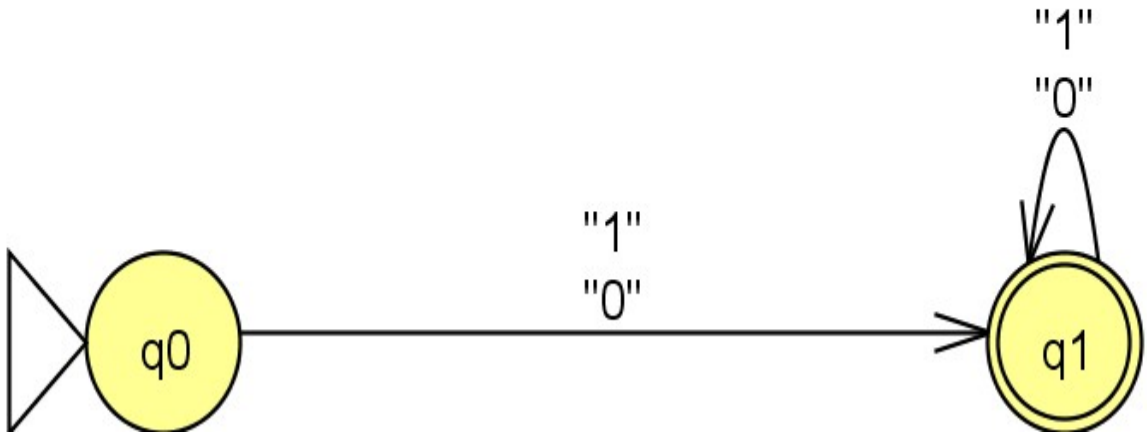
```
initial: 0
final: 1
(0, "I") -> 1  Ação semântica: gera (lambda (x) x)
(0, "K") -> 1  Ação semântica: gera (lambda (x y) x)
(0, "k") -> 1  Ação semântica: gera (lambda (x y) x)
(0, "S") -> 1  Ação semântica: gera (lambda (x y z) ((x z) (y z)))
(0, "s") -> 1  Ação semântica: gera (lambda (x y z) ((x z) (y z)))
(0, NonemptyJotExpr) -> 1  Ação semântica: Nenhuma
(0, quoteExp) -> 1  Ação semântica: Nenhuma
(0, IotaExpr) -> 1
(0, "(") -> 2       Ação semântica: Gera (
(2, CCExpr) -> 3    Ação semântica: Nenhuma
(3, ")") -> 1       Ação semântica: Gera )
```



NonemptyJotExpr

Marcação de estados:
NonemptyJotExpr = 0 (0 *"0"* 2 | *0 "1" 3) 1 { 4 "0" 5* | *4 "1"* 6 } 4 .

```
initial: 0
final: 1
(0, "0") -> 1  Ação semântica:
(0, "1") -> 1  Ação semântica:
(1, "0") -> 1  Ação semântica:
(1, "1") -> 1  Ação semântica:
```



2a Parte da P2 - Definição de características do compilador

O que foi definido para o compilador:

1 - Realizar a tradução de código escrito em LazyK para a notação lambda (ou seja, I é traducido para (lambda(x) x))
2 - Gerar um arquivo com código em C que executa o código traduzido para a notação Lambda.
3 - Como primitiva do ambiente de execução foi definida uma função em C com a assinatura:

```
void  executaCalculoLambda(char str[])
```

Esta função é responsável por executar o código em notação lambda gerado.

Para exemplificar, suponha que o compilador receba como entrada o código I.

O compilador primeiro traduz I=> (lambda(x)x) e em seguida gera o código:

```
#include <stdio.h>

int main(void)
{

executaCalculoLambda("(lambda(x)x)");
system("pause");
return 0;
}
```

Como não dispusemos de uma função executaCalculoLambda exatamente como a que desejamos, optamos por escrever essa função apenas exibindo na tela o código lambda a ser calculado.

3a Parte da P2 - Codificação

O compilador foi escrito reutilizando o compilador já desenvolvido para esta disciplina.

Os automatos das submáquinas que esecutam a análise sintática foram criados utilizando o metacompilador online desenvolvido por Fabio Yamate (http://radiant-fire-72.heroku.com/)

As descrições dos automatos foram adaptadas para o formato aceito pelo nosso metacompilador.

O nosso metacompilador está na classe Util.java e é executado pelos métodos

public static Integer obterEstadoInicial(String strTransicoes)

public static Integer[] obterEstadosFinais(String strTransicoes)

public static List obterTransicoes(String strTransicoes)

Estes métodos são chamados pela classe Maquina.java , a partir do método

public void inicializarMaquina(int quantidadeDeEstados,String strTransicoes, TipoMaquina tipoMaquina)

Este método de inicialização é, por sua vez, chamado a partir do contrutor da classe Maquina.java

Este é um exemplo de chamada do construtor para a submaquina Expr2, executado na classe Sintatico.java

```
String strExpr2 = "initial: 0%"+
                  "final: 1%"+
                  "(0, \"I\") -> 1%"+
                  "(0, \"K\") -> 1%"+
                  "(0, \"k\") -> 1%"+
                  "(0, \"S\") -> 1%"+
                  "(0, \"s\") -> 1%"+
                  "(0, NonemptyJotExpr) -> 1%"+
                  "(0, quoteExp) -> 1%"+
                  "(0, IotaExpr) -> 1%"+
                  "(0, \"(\") -> 2%"+
                  "(2, Expr) -> 2%"+
                  "(2, \")\") -> 1%";

   Maquina MaquinaExpr2= new Maquina(TipoMaquina.EXPR2,strExpr2,11);
```

O método inicializarMaquina cria uma tabela de transição a partir da lista de transições obtidas pelo metacompilador

A execução do sintatico se baseia nos seguintes passos:

1. O construtor instancia as submaquinas com o uso do metacompilador, criando as respectivas tabelas de transição
2. Inicialmente uma pilha é carregada com a máquina Program, de mais alto nível e seu estado inicial

3. O seguinte laço é executado até a pilha ser esvaziada:
    a. Pega o próximo token.
    b. Obtém o proximo passo da máquina atualmente em execuçao a partir do estado atual e do próximo token
    c. Obtém a próxima máquina que deve ser executada
    d. Verifica se é necessário transitar para uma nova máquina
    e. Verifica se é necessário retornar para a máquina anterior
    f. Obtém a identificação da ação semantica a ser executada
    g. Chama o semantico para executar a ação semantica
        i. O semantico executa uma ação semantica caso ela exista e gera código quando possível
    h. Se precisa retornar a máquina, faz o pop(maquina anterior, proximo estado) na pilha e recupera a máquina e o estado de execuçao anterior
    i. Se precisa transitar para uma nova máquina, pas o push(maquina atual, proximo estado) na pilha e transita para o estado inicial da próxima máquina
    j. Caso contrario, atualiza o próximo estado e le o próximo token
4. Quando a pilha é esvaziada é porque o código acabou
    a. Se ainda existem tokens na lista de tokens, um erro sintático é gerado
5. O código final é atualizado e o arquivo com o código final é criado.